CAPITOLO 5

Thread e multithreading

di Eric Williams

IN QUESTO CAPITOLO

Multithreading

Una caratteristica che fa di Java un linguaggio potente è il supporto per la programmazione multithreading integrato nel linguaggio, cosa unica in quanto la maggior parte dei linguaggi moderni non includono il multithreading oppure lo forniscono in modo non integrato. Al contrario, Java offre una visuale unica e integrata del multithreading.

Questo è un aspetto essenziale della programmazione in Java; per essere padroni del linguaggio, è necessario innanzitutto acquisire familiarità con i concetti del multithreading, per poi apprendere il modo in cui la programmazione multithreading e concorrente viene eseguita in Java.

Questo capitolo presenta un'introduzione completa ai thread di Java, trattando i seguenti argomenti.

- ✔ Come scrivere e iniziare thread personalizzati.
- ✔ Un riepilogo completo sulle classi `Thread` e `ThreadGroup`.
- ✔ Come rendere le classi sicure rispetto ai thread.
- ✔ Un'introduzione ai monitor di Java.
- ✔ Come coordinare le azioni di diversi thread.

Per la maggior parte dei programmatori Java, il multithreading e la programmazione concorrente sono concetti non familiari. Se si conoscono solo linguaggi a thread singolo, come Visual Basic, Delphi, Pascal, Cobol e così via, si potrebbe pensare che i thread siano troppo difficili da apprendere, invece il modello è semplice e facile da capire. I thread sono un aspetto quotidiano dello sviluppo di applicazioni e di applet Java.

Che cos'è un thread?

Agli albori dell'era dei computer esistevano solo computer *monotasking*, in grado cioè di eseguire un solo compito alla volta. Quei computer grossi e ingombranti iniziavano un lavoro, lo portavano a termine, quindi ne iniziavano uno nuovo e così via. Quando i progettisti cominciarono a sentirsi insoddisfatti di questi sistemi, detti *batch*, riscrissero i programmi che facevano funzionare i computer e in questo modo nacque il moderno sistema operativo multitasking.

Il termine *multitasking* si riferisce alla capacità di un computer di eseguire contemporaneamente diversi compiti (*task*). La maggior parte dei sistemi operativi moderni, come Windows 95 o Solaris, può eseguire contemporaneamente due o più programmi: mentre si utilizza Netscape per prelevare un file di grandi dimensioni, è possibile giocare al Solitario in una finestra diversa; entrambi i programmi vengono eseguiti contemporaneamente.

Il *multithreading* è un'estensione del paradigma del multitasking: a differenza di questo, che agisce su diversi programmi, il multithreading agisce su diversi *thread* all'interno di un singolo programma e così non solo il sistema operativo esegue diversi programmi, ma ogni programma può eseguire diversi thread al suo interno. Ad esempio, utilizzando un browser Web è possibile stampare una pagina, prelevarne un'altra e compilare un modulo, il tutto contemporaneamente.

Un *thread* è una singola sequenza di esecuzione in un programma. Finora sono state descritte applicazioni monothreading, come ad esempio la seguente:

```
class MainInUnUnicoThread {
    public static void main(String[] args) {
        // main() è eseguito in un singolo thread
        System.out.println(Thread.currentThread());
        for (int i=0; i<1000; i++) {
            System.out.println("i == " + i);
        }
    }
}
```

Questo esempio è semplicistico, ma mostra l'utilizzo di un singolo thread in Java. Quando inizia un'applicazione, la macchina virtuale (VM) esegue il metodo `main()` all'interno di un thread di Java. In questo singolo thread, il metodo `main()` di questa semplice applicazione conta da 0 a 999, stampando di volta in volta il valore.

La programmazione all'interno di una singola sequenza di controllo può limitare la possibilità di produrre software Java utilizzabile (si immagini un sistema operativo in grado di eseguire un solo programma alla volta o un browser Web in grado di prelevare una singola

pagina alla volta). Quando si scrive un programma, spesso si desidera che esso faccia diverse cose contemporaneamente, ad esempio richiamare un'immagine dalla rete mentre si richiede una relazione aggiornata sul mercato azionario e si eseguono diverse animazioni. Questo è il tipo di situazione in cui risultano utili i thread di Java. Ogni thread rappresenta una sequenza di controllo che viene eseguita in modo indipendente. Un thread può scrivere un file sul disco, mentre un altro risponde agli eventi della tastiera generati dall'utente.

Prima di passare ai dettagli dei thread di Java, si osservi come si presenta un'applicazione multithreading. Il Listato 5.1 modifica la precedente applicazione monothreading per sfruttare i vantaggi offerti dai thread. Invece di contare da 0 a 999 in un unico thread, questa applicazione utilizza cinque thread diversi, ognuno dei quali conta 200 numeri: da 0 a 199, da 200 a 399 e così via. Non è necessario capire adesso i dettagli di questo esempio, che viene presentato solo per introdurre i thread.

Listato 5.1 *Una semplice applicazione multithreading.*

```
class CountThreadTest extends Thread {
    int from, to;
    public CountThreadTest(int from, int to) {
        this.from = from;
        this.to = to;
    }
    // il metodo run() è come main() per un thread
    public void run() {
        for (int i=from; i<to; i++) {
            System.out.println("i == " + i);
        }
    }
    public static void main(String[] args) {
        // avvia 5 thread, ognuno dei quali conta 200 numeri
        for (int i=0; i<5; i++) {
            CountThreadTest t = new CountThreadTest(i*200, (i+1)*200);
            // avviando un thread si fa partire una sequenza di controllo
            // separata e si esegue il metodo run() del thread
            t.start();
        }
    }
}
```

Quando si avvia questa applicazione, la VM richiama il metodo `main()` nel proprio thread. `main()` quindi avvia cinque thread separati per eseguire l'operazione di conteggio. Nella Figura 5.1 sono mostrati i thread dell'applicazione `CountThreadTest`.

Nonostante possa sembrare che diversi thread eseguano i compiti contemporaneamente, tecnicamente ciò non è vero. Ancora oggi, la maggior parte dei computer ha un unico processore e questi computer possono eseguire al massimo un compito alla volta. Nei sistemi con un unico processore, il sistema operativo passa continuamente da un compito o da un thread all'altro, permettendo a ogni compito o thread di utilizzare la CPU per un breve periodo. Questo argomento è discusso dettagliatamente nel paragrafo: "Gestione dei thread" più avanti in questo capitolo.

Figura 5.1 *Thread di Java in parallelo.*

I thread di Java

Il supporto di diversi thread di esecuzione non è un'invenzione di Java. I thread esistono da diverso tempo e sono stati implementati in molti linguaggi di programmazione. Tuttavia, i programmatori hanno dovuto faticare molto a causa della mancanza di standard per i thread; piattaforme diverse hanno package di thread diversi, ognuno con un'API diversa; i sistemi operativi non supportano i thread in modo uniforme, alcuni li supportano nel nucleo, altri no. Solo recentemente è emerso uno standard per i thread, POSIX (standard IEEE 1003.1c-1995). Tuttavia, lo standard dei thread POSIX definisce solo un'interfaccia di programmazione C, non un'interfaccia Java, e non è ancora implementato in modo diffuso.

Uno dei vantaggi principali di Java è che presenta ai programmatori un'API multithreading unificata, supportata da tutte le macchine virtuali di Java su tutte le piattaforme. Quando si utilizzano i thread di Java, non ci si deve preoccupare di quali package di thread sono disponibili nella piattaforma sottostante, o se il sistema operativo supporta i thread nel nucleo. La macchina virtuale isola il programmatore dai dettagli dei thread specifici della piattaforma e l'API dei thread è identica per tutte le implementazioni di Java.

Creazione di nuovi thread

La prima cosa da sapere sui thread è come crearli e come eseguirli. Questo processo è suddiviso in due passaggi: scrivere il codice che viene eseguito nel thread e scrivere il codice che avvia il thread.

Nei programmi monothreading, quando si scrive una funzione `main()`, il metodo viene eseguito in un singolo thread. La macchina virtuale di Java offre un ambiente multithreading, ma avvia le applicazioni utente richiamando `main()` in un unico thread.

Il metodo `main()` di un'applicazione fornisce la logica centrale per il thread principale dell'applicazione stessa; scrivere il codice per un thread è come scrivere `main()`: è necessario fornire un metodo che implementa la logica principale del thread; questo metodo si chiama sempre `run()` e ha la seguente segnatura:

```
public void run();
```

A differenza del metodo `main()`, che è statico perché un'applicazione inizia con un unico `main()`, il metodo `run()` non è statico. Un'applicazione può avere tuttavia diversi thread e pertanto la logica principale per un thread è associata a un oggetto, l'oggetto `Thread`.

È possibile fornire l'implementazione di `run()` in due modi diversi: Java supporta il metodo `run()` nelle sottoclassi della classe `Thread` e anche tramite l'interfaccia `Runnable`.

Sottoclassi della classe Thread

In questo paragrafo viene descritto come creare un nuovo thread tramite sottoclassi di `java.lang.Thread`.

Si inizia con una situazione plausibile in cui potrebbe essere utile un thread. Si supponga di costruire un'applicazione, in una parte della quale è necessario copiare un file da una directory a un'altra. Quando si esegue l'applicazione, si scopre che, se il file è di grandi dimensioni, mentre viene copiato l'applicazione rimane in sospeso.

Si stabilisce che la causa è che durante la copia del file l'applicazione non è in grado di rispondere agli eventi dell'interfaccia utente.

Per risolvere questo problema, si decide che l'operazione di copia del file dovrebbe essere eseguita in modo concorrente, in un thread separato; così si crea una sottoclasse della classe `Thread` che contiene questa logica, implementata nel metodo `run()`.

La classe `FileCopyThread` del Listato 5.2 contiene questa logica.

Listato 5.2 *La logica "copia file" in FileCopyThread.*

```
// sottoclasse di Thread per fornire un tipo particolare di thread
class FileCopyThread extends Thread {
    private File from;
    private File to;
    public FileCopyThread(File from, File to) {
        this.from = from;
        this.to = to;
    }
    // implementa the la logica principale del thread nel metodo run()
    // il metodo run() equivale al metodo main() di un'applicazione
    public void run() {
        FileInputStream in = null;
        FileOutputStream out = null;
        byte[] buffer = new byte[512];
        int size = 0;
        try {
            // apre i dispositivi di input e output
            in = new FileInputStream(from);
            out = new FileOutputStream(to);
            // copia 512 byte per volta fino a EOF
            while ((size = in.read(buffer)) != -1) {
               out.write(buffer, 0, size);
            }
        } catch(IOException ex) {
```

```
                ex.printStackTrace();
            } finally {
                // chiude i dispositivi di input e output
                try {
                    if (in != null) { in.close(); }
                    if (out != null) { out.close(); }
                } catch (IOException ex) {
                }
            }
        }
    }
```

Analizzando la classe FileCopyThread, si nota che essa è una sottoclasse di Thread che ha ereditato lo stato e i comportamenti di un Thread, vale a dire la proprietà di "essere un thread."

La classe FileCopyThread implementa la logica principale del thread nel metodo run() che, come detto, è il metodo iniziale per un thread di Java, esattamente come main() è il metodo iniziale per un'applicazione. All'interno di run(), il file di input viene copiato nel file di output in spezzoni da 512 byte. Quando viene creata e avviata un'istanza di FileCopyThread, l'intero metodo run() viene eseguito in un'unica sequenza di controllo separata (più avanti viene spiegato come).

Ora che si sa come si scrive una sottoclasse di Thread, è necessario imparare a utilizzare questa classe come sequenza di controllo separata all'interno di un programma. Per utilizzare un thread, è necessario avviare l'esecuzione concorrente del thread richiamando il metodo start() dell'oggetto Thread.

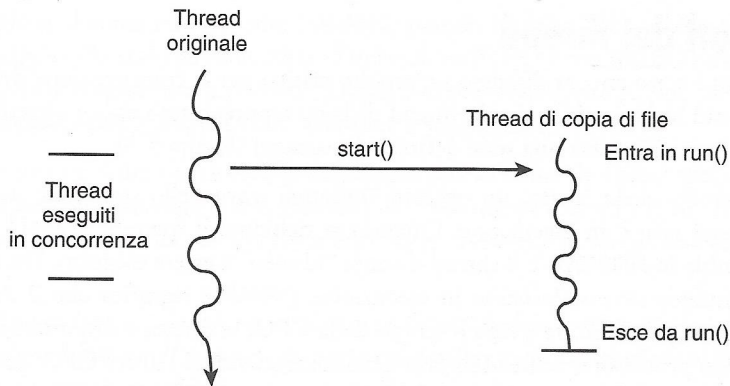
Il seguente codice mostra come si può avviare un'operazione di copia di un file come thread separato:

```
File from = getCopyFrom();
File to = getCopyTo();
// crea un'istanza della classe Thread
Thread t = new FileCopyThread(from, to);
// richiama start() per attivare il thread in modo asincrono
t.start();
```

Richiamando il metodo start() di un oggetto FileCopyThread, inizia l'esecuzione concorrente di quel thread e viene richiamato il suo metodo run(). In questo caso, l'esecuzione della copia del file inizia in modo concorrente con il thread originale. Al termine della copia, il metodo run() restituisce il controllo e termina l'esecuzione concorrente del thread. Nella Figura 5.2 è mostrato questo processo.

Implementazione dell'interfaccia Runnable

A volte non è conveniente creare una sottoclasse di Thread, ad esempio quando si desidera aggiungere un metodo run() a una classe preesistente che non deriva da Thread. Ciò è possibile grazie all'interfaccia Runnable.

Figura 5.2
Copia di file concorrente.

L'API dei thread di Java supporta la nozione di un entità simile a un thread, l'interfaccia `java.lang.Runnable`. `Runnable` è una semplice interfaccia con un unico metodo:

```
public interface Runnable {
    public void run();
}
```

Questa interfaccia dovrebbe essere familiare. Nel paragrafo precedente è stata discussa la classe `Thread`, che anch'essa supportava il metodo `run()`. Per creare una sottoclasse di `Thread`, era stato ridefinito il metodo `Thread run()`; per utilizzare l'interfaccia `Runnable`, è necessario scrivere un metodo `run()` e aggiungere alla classe `implements Runnable`. Per implementare nuovamente `FileCopyThread` dell'esempio precedente come interfaccia `Runnable` sono necessarie alcune modifiche:

```
// L'implementazione di Runnable è un modo diverso di utilizzare i thread
class FileCopyRunnable implements Runnable {
    // il resto della classe rimane per lo più inalterato
    ...
}
```

Per utilizzare un'interfaccia `Runnable` come sequenza di controllo separata, è necessaria la cooperazione di un oggetto `Thread`. Nonostante l'oggetto `Runnable` contenga la logica principale, `Thread` è l'unica classe che incapsula il meccanismo di avvio e di controllo di un thread. Per supportare `Runnable`, è stato aggiunto a diversi costruttori della classe `Thread` un parametro `Runnable` separato. Quando inizia l'esecuzione, un thread inizializzato con un oggetto `Runnable` richiama il metodo `run()` di quell'oggetto.

Di seguito è presentato un esempio di come avviare un thread utilizzando `FileCopyRunnable`:

```
File from = new File("file.1");
File to = new File("file.2");
// crea un'istanza di Runnable
Runnable r = new FileCopyRunnable(from, to);
// crea un'istanza di Thread, passando Runnable
Thread t = new Thread(r);
// avvia il thread
t.start();
```

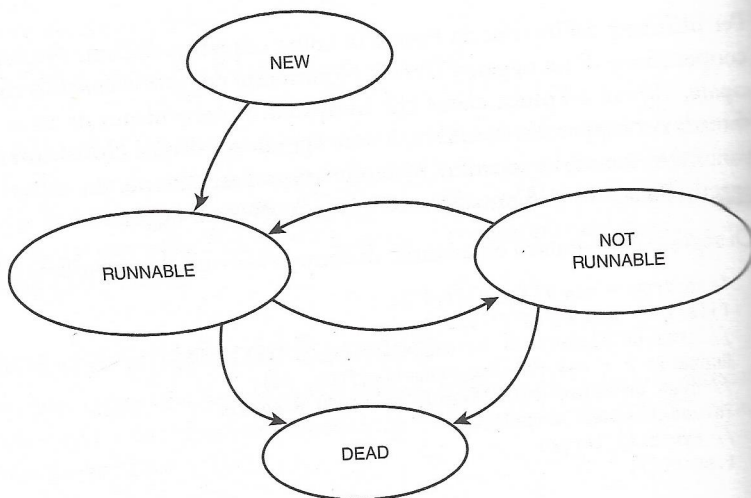
Stati dei thread

Non è stato ancora discusso un aspetto critico per la comprensione del funzionamento dei thread in Java: gli stati. Un thread di Java, rappresentato da un oggetto `Thread`, durante la sua vita attraversa una serie definita di passaggi (Figura 5.3).

Quando viene creato, un oggetto `Thread` si trova nello stato `NEW`; in questo momento il thread non è in esecuzione. Quando si richiama il metodo `start()` del `Thread`, lo stato cambia in `RUNNABLE` e il thread diventa "idoneo" a essere eseguito. Un thread `RUNNABLE` non è tuttavia necessariamente in esecuzione: `RUNNABLE` significa che il thread è vivo e che il sistema può allocare a esso il tempo della CPU, se questa è disponibile. Nei sistemi con un unico processore, i thread di Java devono condividere l'unica CPU; inoltre, anche il task (o processo) della macchina virtuale di Java deve condividere la CPU con gli altri task in esecuzione nel sistema. Più avanti, nel paragrafo: "Gestione e priorità", si discuterà in maggiore dettaglio il modo in cui il tempo di una CPU viene allocato a un thread.

Quando a un thread `RUNNABLE` accadono determinati eventi, il thread può passare allo stato `NOT RUNNABLE`, che significa che il thread è vivo, ma non idoneo per l'esecuzione; al thread non viene allocato il tempo della CPU. Alcuni eventi che possono causare il passaggio di un thread allo stato `NOT RUNNABLE` sono i seguenti.

- ✔ Il thread è in attesa del completamento di un'operazione di I/O.
- ✔ Il thread è stato tenuto in sospeso per un determinato periodo (utilizzando il metodo `sleep()`).
- ✔ È stato richiamato il metodo `wait()` (si veda il paragrafo: "Sincronizzazione" più avanti in questo capitolo).
- ✔ Il thread è stato sospeso (utilizzando il metodo `suspend()`).

Figura 5.3
Stati dei thread.

Un thread NOT RUNNABLE diventa nuovamente RUNNABLE quando termina la condizione che ne ha causato il passaggio allo stato NOT RUNNABLE (l'operazione di I/O è terminata, il thread ha terminato il periodo di sospensione e così via). Durante la sua vita, un thread può passare diverse volte dallo stato RUNNABLE a quello NOT RUNNABLE e viceversa.

Quando termina un thread, si dice che è DEAD (morto). Ciò può avvenire in diversi modi. Di norma, un thread muore quando viene il suo metodo run() restituisce il controllo ma anche quando viene richiamato il suo metodo stop() o destroy(). Non vi è modo di resuscitare un thread DEAD.

Quando un thread muore, tutte le risorse da esso utilizzate, incluso l'oggetto Thread *stesso, diventano disponibili per il recupero da parte del garbage collector, ovviamente se in nessuna altra parte vi sono riferimenti a queste risorse. I programmatori sono responsabili di eliminare le risorse del sistema (chiudere file aperti, eliminare contesti grafici e così via) mentre un thread termina, ma quando un thread muore non è richiesta nessuna di queste operazioni.*

L'API Thread

Nel seguito è presentata un'analisi dettagliata dell'API Thread di Java.

Costruttori

La classe Thread ha sette diversi costruttori:

```
public Thread();
public Thread(Runnable target);
public Thread(Runnable target, String nome);
public Thread(String nome);
public Thread(ThreadGroup gruppo, Runnable target);
public Thread(ThreadGroup gruppo, Runnable target, String nome);
public Thread(ThreadGroup gruppo, String nome);
```

Questi costruttori rappresentano la maggior parte delle combinazioni di tre diversi parametri: *nome*, *gruppo* e un oggetto Runnable *target*. Per capire i costruttori, è necessario comprendere questi tre parametri:

- *nome* è il nome (stringa) da assegnare al thread. Se non si specifica un nome, il sistema ne genera uno unico con la forma Thread-N, dove N è un intero unico.
- *target* è l'istanza di Runnable il cui metodo run() viene eseguito come metodo principale del thread.
- *gruppo* è il ThreadGroup a cui verrà aggiunto questo thread (la classe ThreadGroup viene discussa in dettaglio più avanti in questo capitolo).

La costruzione di un nuovo thread non ne avvia l'esecuzione. Per avviare l'oggetto Thread, è necessario richiamare il suo metodo start().

Quando si crea un thread, la priorità e lo stato di demone vengono impostati sugli stessi valori del thread da cui deriva.

L'allocazione di un thread utilizzando `new Thread()` *è possibile, ma non ha alcuna utilità. Quando si costruisce un thread direttamente, senza creare una sottoclasse, l'oggetto* `Thread` *necessita di un oggetto* `Runnable` *di destinazione, in quanto la classe* `Thread` *non contiene la logica dell'applicazione.*

Attribuzione di nomi

```
public final String getName();
public final void setName(String nome);
```

Ogni thread di Java ha un nome che può essere impostato durante la creazione oppure per mezzo del metodo `setName()`. Se non si specifica un nome durante la creazione, il sistema ne genera uno unico con la forma `Thread-N`, dove `N` è un intero unico. Questo nome può essere modificato successivamente utilizzando `setName()`.

Il nome di un thread può essere richiamato utilizzando il metodo `getName()`.

I nomi dei thread sono importanti, in quanto costituiscono un modo utile per il programmatore per identificare thread particolari durante il debugging; A tale scopo i nomi dovrebbero essere scelti in modo che possano aiutare a identificare lo scopo o la funzione del thread.

Avvio e interruzione

Per avviare e interrompere un thread dopo averlo creato, sono necessari i seguenti metodi:

```
public void start();
public final void stop();
public final void stop(Throwable ogg);
public void destroy();
```

Per avviare un nuovo thread, si crea un nuovo oggetto `Thread` e si richiama il suo metodo `start()`. Se `start()` viene richiamato più di una volta per lo stesso thread, viene generata un'eccezione.

Come discusso nel paragrafo: "Stati dei thread" precedentemente in questo capitolo, vi sono due modi principali in cui un thread può terminare: quando il suo metodo `run()` restituisce il controllo oppure tramite il metodo `stop()` o `destroy()`.

Quando lo si richiama su un thread, il metodo `stop()` termina generando un'eccezione `ThreadDeath`, come avviene quando viene eseguito `throw new ThreadDeath()` all'interno del thread. La differenza consiste nel fatto che `stop()` può essere richiamato anche da altri thread, mentre l'istruzione `throw` influisce solo sul thread corrente.

Per capire i motivi per cui `stop()` è implementato in questo modo, si pensi a che cosa significa interrompere un thread in esecuzione. I thread attivi sono parte di un programma in esecuzione e ogni thread `RUNNABLE` esegue un'operazione. È probabile che ogni thread occupi risorse del sistema: descrittori di file, contesti grafici, monitor (discussi successiva-

mente) e così via. Se l'interruzione di un thread facesse terminare immediatamente tutte le attività relative, queste risorse potrebbero non essere liberate in modo appropriato e il thread potrebbe non avere la possibilità di chiudere i file aperti o di rilasciare i monitor bloccati. Se un thread fosse interrotto nel momento sbagliato, non potrebbe liberare queste risorse, causando possibili problemi alla macchina virtuale (ad esempio potrebbero esaurirsi i descrittori di file aperti).

Per permettere di interrompere i thread in modo corretto, viene data la possibilità di liberare tutte le risorse, generando un'eccezione `ThreadDeath` che si propaga nello stack dei thread e nei gestori di errore che si trovano nello stack, inclusi i blocchi `finally`. Anche i monitor vengono rilasciati da questo processo di "svolgimento" dello stack.

Il Listato 5.3 mostra il modo in cui, richiamando `stop()` su un thread in esecuzione, viene generata un'eccezione `ThreadDeath`.

Listato 5.3 *Creazione di un'eccezione ThreadDeath con stop().*

```
class DyingThread extends Thread {
    // main(), questa classe è un'applicazione
    public static void main(String[] args) {
        Thread t = new DyingThread();          // crea il thread
        t.start();                             // avvia il thread
        // attende un attimo
        try { Thread.sleep(100); } catch (InterruptedException e) { }
        t.stop();                              // ora interrompe il thread
    }
    // run(),questa classe è anche un thread
    public void run() {
        int n = 0;
        PrintStream ps = null;
        try {
            ps = new PrintStream(new FileOutputStream("big.txt"));
            while (true) {                         // per sempre
                ps.println("n == " + n++);
                try { Thread.sleep(5); } catch (InterruptedException e) { }
            }
        } catch (ThreadDeath td) {                 // aspetta stop()
            System.out.println("Sto facendo pulizia.");
            ps.close();                            // chiude il file aperto
            // è molto importante eseguire nuovamente il throw di ThreadDeath
            throw td;
        } catch (IOException e) {
        }
    }
}
```

La classe `DyingThread` è composta da due parti. Il metodo `main()` genera un nuovo `DyingThread`, rimane in attesa per un periodo di tempo, quindi invia uno `stop()` al thread. Il metodo `DyingThread run()`, che viene eseguito nel thread generato, apre un file e vi scrive periodicamente l'output. Quando il thread riceve il metodo `stop()`, intercetta l'eccezione `ThreadDeath` e chiude il file aperto, quindi genera nuovamente l'eccezione `ThreadDeath`.

Quando si esegue il codice del Listato 5.3, si ottiene il seguente output:

```
Sto facendo pulizia.
```

Java fornisce ai programmatori un meccanismo appropriato per scrivere il codice di pulizia, vale a dire il codice che viene eseguito quando avvengono degli errori o quando termina un programma o un thread (la pulizia include la chiusura di file aperti, l'eliminazione di contesti grafici, il nascondere le finestre e così via). Il gestore di eccezioni `catch` *e i blocchi* `finally` *sono buoni punti in cui inserire il codice di pulizia.*
I programmatori utilizzano molti stili diversi per scrivere il codice di pulizia. Alcuni lo inseriscono nei gestori di eccezione `catch(ThreadDeath td)`*, come nel Listato 5.4, altri preferiscono utilizzare gestori di eccezioni* `catch(Throwable t)`*. Entrambi questi metodi sono validi, ma nella maggior parte dei casi la soluzione migliore è quella di scrivere il codice di pulizia in un blocco* `finally`*, che viene eseguito indipendentemente dal fatto che il gestore di errori abbia terminato l'esecuzione a causa di un'eccezione generata. Se è stata generata un'eccezione, questa viene generata di nuovo automaticamente dopo che è stato completato il blocco* `finally`*.*

Nonostante la soluzione di `ThreadDeath` conferisca all'applicazione un'alta flessibilità, vi sono dei problemi. Catturando un'eccezione `ThreadDeath`, un thread può impedire che `stop()` abbia l'effetto desiderato. Il codice è banale:

```
// impedisce a stop() di funzionare
catch (ThreadDeath td) {
    System.err.println("Prova a fermarmi. Sono invincibile.");
    // oh no, non sono riuscito a effettuare nuovamente il throw di td
}
```

La chiamata a `stop()` non è sufficiente per garantire che un thread termini. Questo è un problema serio per i browser compatibili con Java: non vi è garanzia che un applet termini quando `stop()` viene richiamato su un thread che appartiene all'applet.

Il metodo `destroy()` è più forte del metodo `stop()`; è stato progettato per terminare un thread senza ricorrere al meccanismo di `ThreadDeath`. Esso fa terminare il thread immediatamente, senza che avvenga la pulizia e che vengano rilasciate le risorse trattenute dal thread.

Il metodo `destroy()` *non è implementato in nessuna versione del Java Development Kit, inclusa la 1.1. Richiamando questo metodo si genera un'eccezione* `NoSuchMethodError`*. Nonostante non vi siano stati commenti su quando questo metodo verrà implementato, è probabile che non sarà disponibile finché JavaSoft potrà implementarlo in modo che venga eseguita la pulizia dell'ambiente del thread che viene terminato (monitor bloccati, operazioni di I/O in esecuzione e così via).*

Gestione e priorità

La *gestione* dei thread è definita come il meccanismo utilizzato per determinare il modo in cui il tempo della CPU viene allocato ai thread `RUNNABLE`, vale a dire quando vengono realmente eseguiti per un periodo di tempo nella CPU del computer.

Un meccanismo di gestione dei thread è *preemptive* o *non preemptive.* Nel primo caso, il gestore di thread pone in pausa un thread in esecuzione per permettere che ne vengano eseguiti altri. Un gestore non preemptive non interrompe mai un thread in esecuzione, ma si basa su quest'ultimo per cedere il controllo della CPU, in modo che possano essere eseguiti gli altri thread. Nella gestione non preemptive, se il thread in esecuzione non passa il controllo, gli altri thread potrebbero non ottenere mai il tempo dalla CPU.

All'interno dei gestori di thread classificati come preemptive, vi è un'ulteriore suddivisione: i gestori preemptive possono essere *a suddivisione di tempo* o no. Nel primo caso, il gestore alloca un periodo di tempo durante il quale ogni thread può utilizzare la CPU; quando questo tempo è trascorso, il gestore sostituisce il thread con un altro. Per determinare se sostituire un thread, un gestore che non opera a suddivisione di tempo non utilizza il tempo trascorso, ma altri criteri, quali la priorità e lo stato di I/O.

Sistemi operativi e package di thread diversi implementano politiche di gestione diverse, ma Java è inteso per essere indipendente dalla piattaforma. La correttezza di un programma di Java non dovrebbe dipendere dalla piattaforma su cui viene eseguito, pertanto i progettisti di Java hanno deciso di isolare il programmatore dalla maggior parte dei legami con le piattaforme fornendo un'unica garanzia per la gestione dei thread: il thread `RUNNABLE` con la priorità più alta viene sempre selezionato per l'esecuzione prima dei thread con priorità più bassa. Quando vi sono diversi thread con pari priorità, viene garantita l'esecuzione di uno solo di essi. I thread di Java sono garantiti come preemptive, ma non sempre a suddivisione di tempo. Se un thread con priorità più alta di quella del thread corrente diventa `RUNNABLE`, il gestore sostituisce quest'ultimo con il primo. Tuttavia, se diventa `RUNNABLE` un thread con priorità più bassa o uguale a quella del thread corrente, non vi è garanzia che al nuovo thread verrà allocato il tempo della CPU, finché esso non diventa il thread `RUNNABLE` con la priorità più alta.

L'implementazione corrente della VM di Java utilizza package di thread diversi per ogni piattaforma; in questo modo, il comportamento del gestore di thread di Java varia leggermente da piattaforma a piattaforma. È opportuno verificare con il fornitore della VM di Java se la VM utilizza thread nativi e se i thread nativi della piattaforma sono a suddivisione di tempo (alcuni package di thread nativi, ad esempio quelli per Solaris, non sono a suddivisione di tempo).

Nonostante i thread di Java non siano garantiti come thread a suddivisione di tempo, ciò non dovrebbe costituire un problema per la maggior parte delle applicazioni e degli applet. I thread di Java rilasciano il controllo della CPU quando diventano `NOT RUNNABLE`. Se un thread è in attesa di un I/O, se è in sospeso o se è in attesa di immettersi in un monitor, il gestore di thread ne seleziona uno diverso per l'esecuzione. Di norma, vi possono essere problemi solo con i thread che eseguono calcoli intensivi (senza I/O). Per evitare che vengano eseguiti altri thread, si dovrebbe utilizzare il codice riportato di seguito; un thread simile farebbe morire gli altri thread solo su alcune piattaforme; su Windows NT, ad esempio, sarebbe comunque possibile eseguire altri thread.

```
int i = 0;
while (true) {
    i++;
}
```

Esistono diverse tecniche che è possibile implementare per evitare che un thread occupi troppo tempo della CPU.

- ✔ Non scrivere codice come `while (true) { }`. È accettabile avere cicli infiniti, se ciò che avviene nel ciclo include l'I/O, `sleep()` o il coordinamento tra i thread (utilizzando i metodi `wait()` e `notify()`, discussi più avanti in questo capitolo).
- ✔ Richiamare di tanto in tanto `Thread.yield()` quando si eseguono operazioni che prevedono un utilizzo intensivo della CPU. Il metodo `yield()` permette al gestore di dedicare del tempo all'esecuzione di altri thread.
- ✔ Diminuire la priorità dei thread che prevedono un utilizzo intensivo della CPU. I thread con priorità più bassa vengono eseguiti solo quando quelli con priorità più alta non hanno nulla da fare. Ad esempio, il thread del garbage collector di Java ha bassa priorità e pertanto la garbage collection avviene quando non vi sono thread con priorità più alta che necessitano della CPU, evitando in questo modo di tenere impegnato il sistema inutilmente.

Con queste tecniche, le applicazioni e gli applet potranno essere eseguiti senza problemi su qualsiasi piattaforma di Java.

Impostazione della priorità dei thread

```
public final static int MAX_PRIORITY = 10;
public final static int MIN_PRIORITY = 1;
public final static int NORM_PRIORITY = 5;
public final int getPriority();
public final void setPriority(int newPriority);
```

Ogni thread ha una priorità, che eredita dal thread che lo ha creato e che può essere modificata successivamente utilizzando il metodo `setPriority()`. È possibile ottenere la priorità di un thread utilizzando `getPriority()`.

Vi sono alcune costanti simboliche definite nella classe `Thread` che rappresentano la gamma di valori della priorità: `MIN_PRIORITY`, `NORM_PRIORITY` e `MAX_PRIORITY`. I valori variano da 1 a 10; se si cerca di impostare valori al di fuori di questo intervallo, viene generata un'eccezione.

Ripristino dell'esecuzione di un thread

```
public void interrupt();
public static boolean interrupted();
public boolean isInterrupted();
```

Per inviare a un thread un messaggio affinché riprenda l'esecuzione, si richiama `interrupt()` sul suo oggetto `Thread`. Così facendo si genera nel thread un'eccezione `InterruptedException` e si imposta un flag che può essere controllato dal thread in esecuzione utilizzando il metodo `interrupted()` o `isInterrupted()`. Richiamando `Thread.interrupted()`, lo stato di inter-

ruzione del thread corrente viene controllato e reimpostato su `false` (nelle versioni precedenti alla 1.1, lo stato di interruzione non veniva reimpostato). Richiamando `isInterrupted()` su un oggetto `Thread` (che può essere un thread diverso da quello corrente) lo stato di interruzione del thread viene controllato, ma non modificato. Il metodo `interrupt()` è utile per ripristinare l'esecuzione di un thread dopo un'operazione che lo ha bloccato, ad esempio un'operazione di I/O, `wait()` o un tentativo di immettersi in un metodo `synchronized`.

Il metodo `interrupt()` non è completamente implementato nel JDK 1.0.x. Richiamando `interrupt()` su un thread si imposta il flag `interrupted`, ma non si genera un'eccezione `InterruptedException`, né si termina l'operazione che ha causato il blocco nel thread di destinazione; per determinare se il thread è stato interrotto, è necessario controllare `interrupted()`. Il metodo `interrupt()` è pienamente implementato nella versione 1.1 della macchina virtuale di Java.

Sospensione e ripresa dell'esecuzione dei thread

```
public final void suspend();
public final void resume();
```

A volte è necessario mettere in pausa un thread in esecuzione, cosa che è possibile fare utilizzando `suspend()`. Con questo metodo si garantisce che un thread non venga eseguito. Il metodo `resume()` inverte l'operazione di `suspend()`.

Richiamando `suspend()`, un thread passa nello stato `NOT RUNNABLE`; tuttavia, se si richiama `resume()`, non è garantito che il thread di destinazione diventi `RUNNABLE`, in quanto altri eventi potrebbero aver fatto passare il thread nello stato `NOT RUNNABLE` (o `DEAD`).

Pausa di un thread

```
public static void sleep(long millisecondi);
public static void sleep(long millisecondi, int nanosecondi);
```

Per mettere in pausa il thread corrente per un determinato periodo di tempo, si richiama una delle varianti del metodo `sleep()`. Ad esempio, `Thread.sleep(500)` mette in pausa il thread corrente per mezzo secondo, durante il quale il thread è nello stato `NOT RUNNABLE`. Quando scade il tempo specificato, il thread ritorna `RUNNABLE`.

Nelle versioni 1.0.x e 1.1 del JDK, il metodo `sleep(int millisecondi, int nanosecondi)` utilizza il parametro `nanosecondi` per arrotondare il parametro `millisecondi` al millesimo di secondo più prossimo. Non è ancora supportato il metodo `sleep()` in nanosecondi.

Passaggio a un thread diverso

```
public static void yield();
```

Il metodo `yield()` viene utilizzato per indicare al gestore che è arrivato il momento di passare all'esecuzione di altri thread. Se vi sono molti thread `RUNNABLE` in attesa di essere eseguiti, il metodo `yield()` garantisce il passaggio a un thread `RUNNABLE` diverso solo se questo ha priorità per lo meno uguale a quella del thread corrente.

Attesa del termine di un thread

```
public final void join();
public final void join(long millisecondi);
public final void join(long millisecondi, int nanosecondi);
```

A volte i programmi devono attendere che un thread specifico termini; in questo caso occorre richiamare uno dei metodi `join()` sul suo oggetto `Thread`. Ad esempio:

```
Thread t = new AttendeTermineThread();
t.start();
...              // fa qualcos'altro
t.join();         // attende il termine del thread
```

I due metodi `join()` con i parametri relativi alla durata vengono utilizzati per specificare un tempo limite per l'operazione `join()`. Se il thread non termina entro questo tempo, `join()` restituisce comunque il controllo. Per determinare se è stato raggiunto il tempo limite o se il thread è terminato, si utilizza il metodo `isAlive()` di `Thread`.

`join()` senza parametri rimane per sempre in attesa che il thread termini.

Nelle versioni 1.0.x e 1.1 del JDK, il metodo `join(int millisecondi, int nanosecondi)` *utilizza il parametro* `nanosecondi` *per arrotondare il parametro* `millisecond` *al millesimo di secondo più prossimo. Non è ancora supportato il join in nanosecondi.*

Thread demoni

```
public final boolean isDaemon();
public final void setDaemon(boolean on);
```

Alcuni thread sono intesi come thread di background, cioè forniscono servizi ad altri thread. Questi thread sono detti *demoni*. Quando rimangono vivi solo thread demoni, il processo della macchina virtuale di Java termina.

La macchina virtuale di Java ha almeno un thread demone, quello della garbage collection, che è a bassa priorità e che viene eseguito solamente quando il sistema non ha nient'altro da fare.

Il metodo `setDaemon()` imposta lo stato di demone di un thread, mentre il metodo `isDaemon()` restituisce `true` se un thread è un demone, `false` altrimenti.

Altri metodi dei thread

Il metodo `countStackFrames()` restituisce il numero di frame dello stack (attivazioni di metodi) correntemente attivi nello stack di un thread. Quando viene richiamato questo metodo, il thread deve essere sospeso. La segnatura di questo metodo è:

```
public int countStackFrames();
```

Il metodo `getThreadGroup()` restituisce la classe `ThreadGroup` a cui appartiene un thread. Un thread è sempre parte di una sola classe `ThreadGroup`.

La segnatura di questo metodo è:

```
public final ThreadGroup getThreadGroup();
```

Il metodo `isAlive()` restituisce `true` se su un thread è stato richiamato `start()` e se il thread non è ancora morto. In altre parole, `isAlive()` restituisce `true` se il thread è `RUNNABLE` o `NOT RUNNABLE` e `false` se invece è `NEW` o `DEAD`. La segnatura di questo metodo è:

```
public final boolean isAlive();
```

Il metodo `currentThread()` restituisce l'oggetto `Thread` per la sequenza di esecuzione corrente. La segnatura di questo metodo è:

```
public static Thread currentThread();
```

Il metodo `activeCount()` restituisce il numero di thread nella classe `ThreadGroup` del thread attualmente in esecuzione. La segnatura di questo metodo è:

```
public static int activeCount();
```

Il metodo `enumerate()` restituisce tramite il parametro `tarray` un elenco di tutti i metodi nella classe `ThreadGroup` del thread corrente. La segnatura di questo metodo è:

```
public static int enumerate(Thread tarray[]);
```

Il metodo `dumpStack()` viene utilizzato per il debugging e stampa un elenco metodo per metodo della traccia dello stack per il thread corrente nel flusso di output `System.err`. La segnatura di questo metodo è:

```
public static void dumpStack();
```

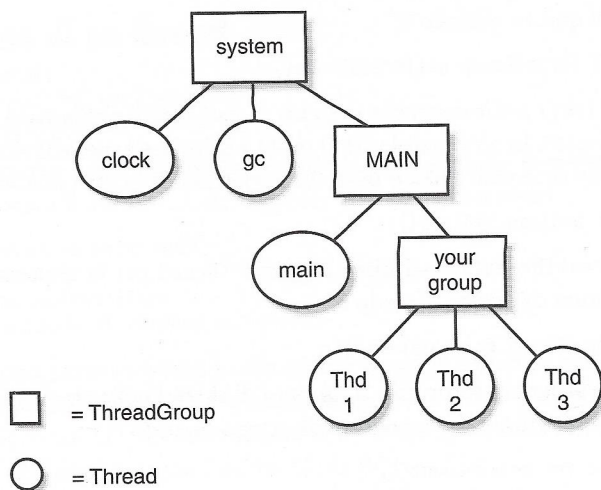
Il metodo `toString()` restituisce una stringa di debugging che descrive un thread. Il prototipo di questo metodo è:

```
public String toString();
```

L'API ThreadGroup

Ogni thread di Java appartiene a una sola istanza di `ThreadGroup`. La classe `ThreadGroup` viene utilizzata per facilitare l'organizzazione e la gestione di gruppi di thread simili. Ad esempio, un gruppo di thread può essere utilizzato da un browser Web per raggruppare tutti i thread che appartengono a un unico applet. Per gestire un intero gruppo di thread che appartiene all'applet è possibile utilizzare comandi unici.

Gli oggetti `ThreadGroup` formano una struttura ad albero; i gruppi possono contenere thread e altri gruppi. Il gruppo di thread superiore è chiamato `system` e contiene diversi thread a livello di sistema, ad esempio il thread del garbage collector e anche l'oggetto `ThreadGroup` principale; il gruppo principale contiene un `Thread` principale, che è il thread in cui viene eseguito `main()`. La Figura 5.4 rappresenta graficamente la struttura di `ThreadGroup`.

Figura 5.4
L'albero gerarchico di ThreadGroup.

Costruttori

La classe ThreadGroup ha due costruttori. Per entrambi è necessario specificare il nome del nuovo gruppo di thread. Uno dei costruttori fa riferimento al gruppo genitore del nuovo ThreadGroup, mentre il costruttore che non ha il parametro *genitore* utilizza come genitore del nuovo gruppo il gruppo del thread correntemente in esecuzione.

```
public ThreadGroup(String nome);
public ThreadGroup(ThreadGroup genitore, String nome);
```

All'inizio il nuovo oggetto ThreadGroup non contiene thread o altri gruppi di thread.

I metodi ausiliari dei thread

La classe ThreadGroup contiene alcuni metodi che operano sui thread all'interno del gruppo. Questi metodi sono detti *ausiliari* e richiamano il metodo di Thread con lo stesso nome per tutti i thread all'interno del gruppo, in modo ricorsivo.

```
public final void suspend();
public final void resume();
public final void stop();
public final void destroy();
```

I metodi ausiliari comprendono suspend(), resume(), stop() e destroy(). Di seguito viene presentato un esempio che mostra come arrestare un intero gruppo di thread con un'unica chiamata di metodo:

```
ThreadGroup group = new ThreadGroup("thread client");
while (una_condizione) {
    Thread t = new Thread(group);
    t.start();
    ...
}
```

```
...
if (kill_em_all) {    // ferma tutti i thread
    group.stop();
}
```

Gli altri metodi ausiliari dei gruppi di thread vengono richiamati in modo simile.

Priorità

Le strutture ThreadGroup semplificano la gestione della priorità dei thread. Dopo aver richiamato setMaxPriority() su un oggetto ThreadGroup, nessun thread all'interno della struttura del gruppo può utilizzare setPriority() per impostare una priorità più alta del valore massimo specificato (non sono influenzate le priorità già impostate dei thread del gruppo).

```
public final int getMaxPriority();
public final void setMaxPriority(int pri);
```

Il metodo getMaxPriority() restituisce il valore massimo della priorità di questa struttura ThreadGroup.

Navigazione nella struttura ThreadGroup

Ogni gruppo di thread può contenere sia thread che gruppi di thread. I metodi activeCount() e activeCountGroup() restituiscono rispettivamente il numero di thread e di gruppi di thread contenuti. Le segnature di questi metodi sono:

```
public int activeCount();
public int activeGroupCount();
```

Il metodo activeCount() restituisce il numero di thread che sono membri di una struttura ThreadGroup (ricorsivamente).

Il metodo activeCountGroup() restituisce il numero di ThreadGroup che sono membri di una struttura ThreadGroup (ricorsivamente).

I seguenti metodi enumerate() possono essere utilizzati per richiamare l'elenco di thread o di gruppi in questo oggetto ThreadGroup:

```
public int enumerate(Thread list[]);
public int enumerate(Thread list[], boolean ricorsione);
public int enumerate(ThreadGroup elenco[]);
public int enumerate(ThreadGroup elenco[], boolean ricorsione);
```

Il parametro *ricorsione*, se true, fa sì che vengano richiamati tutti i thread o i gruppi all'interno di una struttura ThreadGroup (ricorsivamente). Se *ricorsione* è false, vengono richiamati solo i thread o i gruppi nell'oggetto ThreadGroup immediato. I metodi enumerate() senza il parametro *ricorsione* vengono eseguiti nello stesso modo dei metodi enumerate() con *ricorsione* impostato su true.

Il metodo parentOf() restituisce true se un gruppo di thread è il genitore del gruppo specificato, diversamente restituisce false.

La sintassi di questo metodo è:

```
public final boolean parentOf(ThreadGroup g);
```

Il metodo `getParent()` restituisce il nome del genitore di un gruppo di thread oppure `null` se il `ThreadGroup` è quello di livello superiore. La sintassi di questo metodo è:

```
public final ThreadGroup getParent();
```

Il metodo `list()` stampa le informazioni sul debugging relative alla struttura di un `ThreadGroup` (thread e gruppi) in `System.out`. La sintassi di questo metodo è:

```
public void list();
```

Altri metodi di ThreadGroup

Il metodo `getName()` restituisce il nome di un gruppo di thread. La sintassi di questo metodo è:

```
public final String getName();
```

Come per i thread, è possibile definire alcuni gruppi di thread come *demoni*. Se un oggetto `ThreadGroup` è stato impostato come gruppo demone richiamando `setDaemon(true)`, il gruppo viene distrutto dopo che ne sono stati rimossi tutti i thread e tutti i gruppi.

```
public final boolean isDaemon();
public final void setDaemon(boolean daemon);
```

Il metodo `isDaemon()` restituisce `true` se un gruppo di thread è un demone, diversamente restituisce `false`.

Il metodo `toString()` restituisce le informazioni di debugging relative a un gruppo di thread. La sintassi di questo metodo è:

```
public String toString();
```

Quando un thread termina perché non ha catturato un'eccezione, viene richiamato il metodo `uncaughtException()` del gruppo del thread utilizzando come parametri l'oggetto `Thread` e l'eccezione (`Throwable`):

```
public void uncaughtException(Thread t, Throwable e);
```

Il comportamento predefinito di `uncaughtException()` è quello di passare il thread e l'eccezione al genitore di questo gruppo di thread. Il gruppo di thread `system`, se raggiunto, richiama il metodo `printStackTrace()` dell'eccezione `Throwable`, copiando la traccia dello stack dell'eccezione in `System.err`.

Funzionalità di sicurezza

I thread e i gruppi di thread sono considerati risorse critiche del sistema che devono essere protette dalle funzionalità di sicurezza di Java. L'implementazione precisa della politica di sicurezza dipende dall'ambiente. Quando si esegue un'applicazione Java, non vi è sicurezza, a meno che non si installi un `SecurityManager` utilizzando `System.setSecurityManager()`.

Gli applet, tuttavia, utilizzano il `SecurityManager` installato dall'ambiente del browser. Quando si esegue un applet in Netscape Navigator 3.0, ad esempio, a esso è consentito modificare solo i thread e i gruppi di thread creati dall'applet corrente; se si cerca di modificare altri thread o gruppi di thread si genera un'eccezione `SecurityException`.

La sicurezza per la classe `Thread` (come implementata dall'oggetto `SecurityManager` corrente) è implementata per i seguenti metodi:

- ✔ `Thread(ThreadGroup` *`gruppo`*`)`
- ✔ `Thread(ThreadGroup` *`gruppo`*`, Runnable` *`target`*`, String` *`nome`*`)`
- ✔ `Thread(ThreadGroup` *`gruppo`*`, String` *`nome`*`)`
- ✔ `stop()`
- ✔ `suspend()` e `resume()`
- ✔ `setPriority()`
- ✔ `setName()`
- ✔ `setDaemon()`

La sicurezza per la classe `ThreadGroup` (come implementata dall'oggetto `SecurityManager` corrente) è implementata per i seguenti metodi:

- ✔ `ThreadGroup(ThreadGroup` *`genitore`*`, String` *`nome`*`)`
- ✔ `setDaemon()`
- ✔ `setMaxPriority()`
- ✔ `stop()`
- ✔ `suspend()` e `resume()`
- ✔ `destroy()`

Concorrenza

Una delle funzionalità più potenti del linguaggio di programmazione Java è la possibilità di eseguire diversi thread di controllo. L'esecuzione di diversi compiti in parallelo sembra naturale dal punto di vista dell'utente, che può ad esempio trasferire un file da Internet, eseguire un calcolo in un foglio elettronico e stampare un documento nello stesso tempo. Dal punto di vista del programmatore, tuttavia, la gestione della concorrenza non è così naturale come sembra. La concorrenza richiede al programmatore di prendere delle precauzioni speciali per assicurarsi che l'accesso agli oggetti di Java avvenga in modo sicuro rispetto ai thread.

Non vi è nulla nei thread che renda i programmi non sicuri, ma nonostante ciò questi programmi possono essere esposti a situazioni pericolose, a meno che non si prendano misure appropriate.

Il seguente esempio dimostra come un programma con thread possa essere *non sicuro*:

```
public class Contatore {
    private int conta = 0;
    public int incr() {
        int n = conta;
        conta = n + 1;
        return n;
    }
}
```

Per quanto concerne le classi in Java, la classe `Contatore` è semplice, in quanto ha un solo attributo e un solo metodo. Come indica il nome, questa classe viene utilizzata per eseguire un conteggio, ad esempio del numero di volte che un pulsante viene scelto o che l'utente visita un determinato sito Web. Il metodo `incr()`, che restituisce e incrementa il valore corrente del contatore, è il fulcro della classe, ma presenta un problema: in un ambiente multithreading è origine di un comportamento imprevedibile.

Si consideri una situazione in cui un programma di Java ha due thread eseguibili, i quali stanno entrambi per eseguire la seguente riga di codice (che influisce sullo stesso oggetto `Contatore`):

```
int cnt = contatore.incr();
```

Il programmatore non è in grado di conoscere o di controllare l'ordine in cui vengono eseguiti questi due thread. Il gestore di thread di Java ha la completa autorità sulla gestione dei thread. Non vi sono garanzie su quando un thread riceve il tempo della CPU, quando viene eseguito o per quanto tempo. Un thread può essere interrotto dal gestore in qualsiasi momento (si ricordi che il gestore di thread di Java è di tipo preemptive). In un computer con più processori, entrambi i thread possono essere eseguiti in modo concorrente su processori diversi. Nella Tabella 5.1 è descritta una possibile sequenza di esecuzione dei due thread. In questo scenario, il primo thread viene eseguito fino al completamento di `contatore.incr()`, quindi il secondo thread fa lo stesso. Non ci sono sorprese: il primo thread aumenta il valore di `Contatore` a `1` e il secondo thread lo incrementa a `2`.

Tabella 5.1 *Contatore: scenario 1.*

Thread 1	*Thread 2*	*Conteggio*
`cnt = Contatore.incr();`	—	0
`n = conta;      // 0`	—	0
`conta = n + 1;  // 1`	—	1
`return n;       // 0`	—	1
—	`cnt = Contatore.incr();`	1
—	`n = conta;      // 1`	1
—	`conta = n + 1;  // 2`	2
—	`return n;       // 1`	2

Nella Tabella 5.2 è descritta una sequenza di esecuzione diversa. In questo scenario, il primo thread è interrotto da un *cambio di contesto* (il passaggio a un thread diverso) durante l'esecuzione del metodo incr(). Il primo thread rimane temporaneamente in sospeso, mentre viene permesso al secondo di proseguire. Il secondo thread esegue la chiamata al metodo incr(), incrementando il valore di Contatore a 1. Quando viene ripreso il primo thread, si evidenzia un problema: il valore di Contatore non è aggiornato a 2, come ci si aspetterebbe, ma è impostato nuovamente su 1.

Tabella 5.2 *Contatore: scenario 2.*

Thread 1	*Thread 2*	*Conteggio*
`cnt = Contatore.incr();`	—	0
`n = conta;        // 0`	—	0
—	`cnt = Contatore.incr();`	0
—	`n = conta;        // 0`	0
—	`conta = n + 1;  // 1`	1
—	`return n;        // 0`	1
`conta = n + 1; // 1`	—	1
`return n;        // 0`	—	1

Esaminando il thread 1 nella Tabella 5.2, si può vedere una sequenza di operazioni problematica. Dopo l'entrata nel metodo incr(), il valore dell'attributo conta (0) viene memorizzato in una variabile locale n. Il thread viene quindi sospeso per un periodo di tempo, durante il quale viene eseguito un thread diverso (si noti che in questo periodo l'attributo conta viene modificato dal secondo thread). Quando viene ripresa l'esecuzione del thread 1, questo memorizza nuovamente il valore n + 1 (1) nell'attributo conta. Purtroppo non si tratta più del valore corretto per il contatore, in quanto questo era già stato incrementato a 1 dal thread 2.

Il problema della Tabella 5.2 è detto *condizione di conflitto*: il risultato è determinato dall'ordine in cui il tempo della CPU viene allocato ai thread. Di norma si considera non appropriato permettere che tale condizione influisca sul risultato di un programma. Si consideri uno strumento medico che controlla la pressione sanguigna di un paziente; se questo strumento fosse condizionato da conflitti nel software, potrebbe determinare una lettura errata per il medico, che baserebbe le sue decisioni su dati errati. Tutti i programmi multithreading, anche quelli di Java, possono essere influenzati dalle condizioni di conflitto. Per fortuna, Java offre al programmatore gli strumenti necessari per gestire la concorrenza: i *monitor*.

I monitor

Molti testi che trattano di computer e di sistemi operativi discutono l'argomento della programmazione concorrente. La concorrenza è stata oggetto di molte ricerche negli ultimi

anni e sono state proposte e implementate molte soluzioni per controllarla. Tra queste vi sono le seguenti:

- le sezioni critiche;
- i semafori;
- la mutua esclusione;
- il blocco dei record di database;
- i monitor.

Java implementa una variante dei monitor.

Il concetto di *monitor* è stato introdotto da C. A. R. Hoare nel 1974 in un articolo pubblicato nelle *Communications of the ACM.* Hoare ha descritto un oggetto con uno scopo speciale, chiamato monitor, che applica il principio della mutua esclusione tra gruppi di procedure (*mutua esclusione* è un modo diverso per dire "un thread alla volta"). Nel modello di Hoare, ogni gruppo di procedure che necessita della mutua esclusione deve essere sotto il controllo di un singolo monitor. Durante l'esecuzione il monitor permette a un solo thread alla volta di eseguire una procedura controllata. Se un altro thread cerca di richiamare una procedura controllata dal monitor, il thread viene sospeso finché il primo thread non ha completato la chiamata.

I monitor di Java rimangono fedeli al concetto originale di Hoare, con alcune variazioni di ordine minore, che non vengono discusse in questo libro. I monitor in Java mettono in atto l'accesso mutuamente esclusivo ai metodi; più precisamente, mettono in pratica l'accesso mutuamente esclusivo ai metodi `synchronized`. La parola chiave `synchronized` è un modificatore opzionale dei metodi; se è inserita prima del tipo restituito e della segnatura, il metodo viene detto "sincronizzato".

In Java ogni oggetto ha un monitor associato. I metodi `synchronized` richiamati su un oggetto utilizzano il monitor dell'oggetto stesso per limitare l'accesso concorrente. Quando viene richiamato un metodo `synchronized` su un oggetto, il monitor viene consultato per determinare se un altro thread stia eseguendo in concorrenza un metodo `synchronized` sull'oggetto. In caso negativo, al thread corrente viene permesso di *accedere* al monitor (accedere a un monitor è sinonimo di *bloccare* il monitor o di *prendere possesso* del monitor). Se un altro thread ha già avuto accesso al monitor, il thread corrente deve aspettare finché l'altro non lo abbandona.

Utilizzando una metafora, i monitor di Java agiscono da custodi di un oggetto. Se viene richiamato un metodo `synchronized`, il custode permette al thread che ha effettuato la chiamata di passare, quindi chiude il cancello. Mentre il thread si trova nel metodo `synchronized`, vengono bloccate le chiamate degli altri thread ai metodi `synchronized` su quell'oggetto. Questi thread si mettono in fila fuori dal cancello, in attesa che il primo thread esca. Quando questo abbandona il metodo `synchronized`, il custode apre il cancello, permettendo a un solo thread di procedere alla chiamata del metodo `synchronized`, e così via per gli altri thread.

In parole semplici, i monitor di Java mettono in atto un approccio alla concorrenza del tipo "uno alla volta", conosciuto anche come *serializzazione*, da non confondersi con la serializzazione degli oggetti, che è la libreria di Java che permette di leggere e scrivere oggetti in un flusso.

I programmatori che hanno già familiarità con la programmazione multithreading in un linguaggio diverso da Java spesso confondono i monitor con le sezioni critiche, invece i monitor di Java non sono come le tradizionali sezioni critiche; la semplice dichiarazione di un metodo come `synchronized` *non implica che questo possa essere eseguito da un solo thread alla volta, come nel caso delle sezioni critiche. I monitor fanno in modo che un solo thread alla volta possa richiamare quel metodo (o qualsiasi metodo* `synchronized` *su un particolare oggetto in un determinato momento). I monitor di Java sono associati agli oggetti, non ai blocchi di codice; due thread possono eseguire contemporaneamente lo stesso metodo sincronizzato, se il metodo viene richiamato su oggetti diversi (con* `a.metodo()` *e* `b.metodo()`*, dove a != b).*

L'esempio `Contatore` del paragrafo precedente viene ora riscritto in modo da sfruttare i monitor, utilizzando la parola chiave `synchronized`:

```
public class Contatore2 {
    private int conta = 0;
    public synchronized int incr() {
        int n = conta;
        conta = n + 1;
        return n;
    }
}
```

Si noti che il metodo `incr()` non è stato modificato, a eccezione dell'aggiunta della parola chiave `synchronized`.

Che cosa succederebbe se questa nuova classe `Contatore2` fosse utilizzata nello scenario presentato nella Tabella 5.2 (condizione di conflitto)? Nella Tabella 5.3 viene presentato il risultato della stessa sequenza di cambi di contesto.

Nella Tabella 5.3, la sequenza di operazioni inizia nello stesso modo dello scenario indicato nella Tabella 5.2. Il thread 1 inizia l'esecuzione del metodo `incr()` dell'oggetto `Contatore2`, ma viene interrotto da un cambio di contesto. In questo esempio, tuttavia, quando il thread 2 cerca di eseguire il metodo `incr()` sullo stesso oggetto `Contatore2`, non riesce ad acquisire il monitor, che è già in possesso del thread 1, e viene bloccato e tenuto in sospeso finché il monitor diventa disponibile. Quando il thread 1 rilascia il monitor, il thread 2 può acquisirlo e continuare l'esecuzione.

La parola chiave `synchronized` è la soluzione implementata in Java per risolvere il problema del controllo di concorrenza. Come si è visto nell'esempio di `Contatore`, la condizione di conflitto potenziale è stata eliminata aggiungendo al metodo `incr()` il modificatore `synchronized`. Tutti gli accessi al metodo `incr()` di un contatore sono stati serializzati aggiungendo la parola chiave `synchronized`. In termini generali, qualsiasi metodo che modifica gli attributi di un oggetto dovrebbe essere sincronizzato.

Tabella 5.3 *Contatore: scenario 2 rivisto.*

Thread 1	*Thread 2*	*Conteggio*
`cnt = Contatore2.incr();`	—	0
(acquisisce il monitor)	—	0
`n = conta;         // 0`	—	0
—	`cnt = Contatore2.incr();`	0
—	(non può acquisire il monitor)	0
`conta = n + 1; // 1`	(bloccato)	1
`return n;         // 0`	(bloccato)	1
(rilascia il monitor)	(bloccato)	1
—	(acquisisce il monitor)	1
—	`n = conta;         // 1`	1
—	`conta = n + 1;  // 2`	2
—	`return n;         // 1`	2
—	(rilascia il monitor)	2

Ci si potrebbe chiedere quando si vedrà un vero oggetto monitor. Sono state divulgate informazioni varie sui monitor, ma non è possibile consultare la documentazione ufficiale per capire che cos'è un monitor e su come accedervi, in quanto i monitor in Java non hanno una posizione ufficiale nelle specifiche del linguaggio e la loro implementazione non è direttamente visibile al programmatore. I monitor non sono oggetti di Java, non hanno attributi o metodi; sono un concetto oltre l'implementazione del multithreading e della concorrenza di Java. Nella versione 1.x della macchina virtuale di Java della Sun è possibile accedere ai monitor a livello del codice nativo; le specifiche dell'Interfaccia nativa di Java 1.1 definiscono due metodi che agiscono sui monitor: `MonitorEnter()` *e* `MonitorExit()`.

Metodi non sincronizzati

I monitor di Java sono utilizzati solamente assieme ai metodi sincronizzati. I metodi che non sono dichiarati come `synchronized` non cercano di acquisire il possesso del monitor di un oggetto prima dell'esecuzione, ignorandolo completamente.

Un solo thread può eseguire un metodo `synchronized` su un oggetto in un determinato momento, ma qualsiasi numero di thread può eseguire i metodi non sincronizzati. In questo modo possono crearsi alcune situazioni particolari, se non si presta attenzione nel decidere quali metodi dovrebbero essere sincronizzati. Si consideri la classe `ContoCor` inclusa nel Listato 5.4.

Listato 5.4 *La classe ContoCor.*

```
class ContoCor {
  private int saldo;
  public ContoCor(int saldo) {
    this.saldo = saldo;
  }
  public synchronized void transfer(int importo, ContoCor destinazione) {
    synchronized (destinazione) {
      this.prelievo(importo);
      Thread.yield();      // forza un cambio di contesto
      destinazione.deposito(importo);
    }
  }
  public synchronized void prelievo(int importo) {
    if (importo > saldo) {
      throw new RuntimeException("Non c'è protezione!");
    }
    saldo -= importo;
  }
  public synchronized void deposito(int importo) {
    saldo += importo;
  }
  public int getSaldo() {
    return saldo;
  }
}
```

I metodi che modificano gli attributi della classe `ContoCor` sono dichiarati come `synchronized`, ma il metodo `getSaldo()` non è sincronizzato. Sembra che questa classe non abbia problemi con le condizioni di conflitto, ma non è vero!

Per capire la condizione di conflitto a cui è soggetta la classe `ContoCor`, si consideri il modo in cui una banca gestisce i conti correnti. Per una banca, l'esattezza dei conti è di massima importanza: se facesse errori nella contabilità o se riportasse informazioni errate, i clienti non sarebbero molto soddisfatti. Per evitare di riportare informazioni errate, una banca può disattivare le richieste di saldo mentre sul conto viene eseguita una transazione. In questo modo si evita che il cliente veda il risultato di una transazione solo parzialmente completa. Nel Listato 5.4, il metodo `getSaldo()` della classe `ContoCor` non è sincronizzato e ciò può creare problemi.

Si considerino due oggetti `ContoCor` e due thread diversi che eseguono delle azioni su questi conti. Un thread esegue un trasferimento di fondi da un conto all'altro, mentre il secondo esegue una richiesta di informazioni sul saldo, come indicato nel seguente codice:

```
public class ProvaTrasfer implements Runnable {
  public static void main(String[] args) {
    ProvaTrasfer xfer = new ProvaTrasfer();
    xfer.a = new ContoCor(100);
    xfer.b = new ContoCor(100);
    xfer.importo = 50;
    Thread t = new Thread(xfer);
```

```
      t.start();
      Thread.yield();     // forza un cambio di contesto
      System.out.println("Interrogazione: ContoCor a ha : L." + xfer.a.getSaldo());
      System.out.println("Interrogazione: ContoCor b ha : L." + xfer.b.getSaldo());
    }
    public ContoCor a = null;
    public ContoCor b = null;
    public int importo = 0;
    public void run() {
      System.out.println("Prima del trasferimento: a ha : L." + a.getSaldo());
      System.out.println("Prima del trasferimento: b ha : L." + b.getSaldo());
      a.transfer(importo, b);
      System.out.println("Dopo il trasferimento: a ha : L." + a.getSaldo());
      System.out.println("Dopo il trasferimento: b ha : L." + b.getSaldo());
    }
  }
```

In questo esempio vengono creati due conti correnti, ognuno con un saldo di 100 lire. Viene iniziato un trasferimento per spostare 50 lire da un conto all'altro. Il trasferimento non è un'operazione che dovrebbe influenzare il saldo totale dei due conti, che dovrebbe rimanere invariato a 200 lire. Se la richiesta del saldo viene eseguita nel momento giusto, tuttavia, è possibile che la somma totale dei due conti possa essere errata. Se questo programma viene *eseguito utilizzando il JDK versione 1.0 per Solaris, viene stampato il seguente risultato:*

```
Prima del trasferimento: a ha : L.100
Prima del trasferimento: b ha : L.100
Interrogazione: ContoCor a ha : L.50
Interrogazione: ContoCor b ha : L.100
Dopo il trasferimento: a ha : L.50
Dopo il trasferimento: b ha : L.150
```

La richiesta di informazioni riporta che il primo conto contiene 50 lire e il secondo 100 lire, che sommati non danno 200! Che cosa è successo alle altre 50 lire? Nulla, a eccezione del fatto che, mentre la richiesta del saldo verificava i conti, questi si trovavano nel processo di trasferimento al secondo conto. Poiché il metodo `getSaldo()` non è sincronizzato, i clienti non avrebbero problemi a eseguire una richiesta di informazioni sui conti durante il trasferimento dei soldi. La mancanza di sincronizzazione potrebbe lasciare perplesso un cliente sul motivo per cui mancano 50 lire; se invece il metodo `getSaldo()` viene dichiarato come `synchronized`, l'applicazione riporta un risultato diverso. Il codice modificato è:

```
public synchronized int getSaldo() {
return saldo;
}
```

La richiesta del saldo viene bloccata finché il trasferimento è completato. Il risultato del programma è modificato:

```
Prima del trasferimento: a ha : L.100
Prima del trasferimento: b ha : L.100
Interrogazione: ContoCor a ha : L.50
Interrogazione: ContoCor b ha : L.150
Dopo il trasferimento: a ha : L.50
Dopo il trasferimento: b ha : L.150
```

Concetti avanzati sui monitor

I monitor sembrano abbastanza semplici da mettere in pratica: è sufficiente aggiungere ai metodi il modificatore synchronized. In realtà, i monitor di per sé sono semplici, ma se li si considera assieme al resto dell'ambiente di programmazione, ci si rende conto che vi sono alcuni argomenti da capire. Nel seguito sono presentati suggerimenti e tecniche che è necessario conoscere per diventare esperti nella programmazione concorrente in Java.

I metodi static synchronized

I metodi dichiarati come synchronized cercano di acquisire il possesso del monitor dell'oggetto di destinazione. Per quanto concerne i metodi static, che non hanno un oggetto associato, le specifiche del linguaggio sono chiare: quando viene richiamato un metodo static synchronized, il monitor acquisito viene detto *per classe*, per indicare che vi è un monitor per ogni classe che regola l'accesso a tutti i metodi static di quella classe. In un determinato momento può essere attivo in una classe un solo metodo static synchronized.

L'istruzione synchronized

Non è possibile utilizzare metodi synchronized su determinati tipi di oggetti. Ad esempio, non è possibile aggiungere nessun metodo agli oggetti array di Java (tanto meno metodi sincronizzati). Per ovviare a questa restrizione, in Java esiste un secondo modo di interagire con il monitor di un oggetto: l'istruzione synchronized, che ha la seguente sintassi:

```
synchronized ( Espressione ) Istruzione
```

L'esecuzione di un'istruzione synchronized ha lo stesso effetto della chiamata a un metodo synchronized: il possesso del monitor di un oggetto viene acquisito prima che possa essere eseguito un blocco di codice. Con l'istruzione synchronized, l'oggetto il cui monitor è a disposizione è l'oggetto risultante dalla *Espressione*, che deve essere un tipo di oggetto, non un tipo elementare come int, double e così via.

Uno degli utilizzi più importanti dell'istruzione synchronized include il controllo dell'accesso agli oggetti array. Il seguente esempio dimostra l'utilizzo di questa istruzione per permettere l'accesso a un array sicuro rispetto ai thread:

```
void safe_lshift(byte[] array, int conta) {
    synchronized(array) {
        System.arraycopy(array, conta, array, 0, array.size - conta);
    }
}
```

Prima di modificare l'array in questo esempio, la macchina virtuale assegna il possesso del monitor dell'array al thread attualmente in esecuzione. Gli altri thread che cercano di acquisire il monitor dell'array vengono obbligati ad aspettare finché è completata l'operazione di copia dell'array. Naturalmente è necessario prestare attenzione, in quanto gli accessi agli array che non sono controllati da un'istruzione synchronized non sono bloccati.

Pubblici o no?

All'interno della comunità di Java è in corso una discussione sul pericolo potenziale di dichiarare gli attributi come `public`. Quando si considera la concorrenza, risulta evidente che gli attributi `public` possono portare ad avere codice non sicuro rispetto ai thread. Il motivo è che, senza il vantaggio della protezione di un metodo `synchronized`, qualsiasi thread può accedere agli attributi `public`. Quando si dichiara un attributo come `public`, si rinuncia al controllo degli aggiornamenti di quell'attributo: qualsiasi programmatore che utilizzi quel codice ha la possibilità di accedere direttamente agli attributi `public`, e quindi di aggiornarli.

Come regola generale, non è opportuno dichiarare gli attributi non finali come `public`. In questo modo non solo si possono introdurre problemi relativi alla sicurezza dei thread, ma può anche diventare difficile modificare e supportare il codice in revisioni successive.

Si noti tuttavia che i programmatori di Java spesso definiscono costanti simboliche immutabili come attributi di classe `public final` (ad esempio `Event.ACTION_EVENT`). Gli attributi dichiarati in questo modo non presentano problemi di sicurezza per i thread, in quanto le condizioni di conflitto valgono solo per gli oggetti i cui valori possono essere modificati.

L'istruzione `synchronized` è utile anche quando si modificano direttamente le variabili `public` di un oggetto, come nel seguente esempio:

```
void chiama_metodo(UnaClasse obj) {
    synchronized(obj) {
        obj.variable = 5;
    }
}
```

Quando non utilizzare la parola chiave synchronized

Ora si dovrebbe essere in grado di scrivere codice a thread protetto utilizzando la parola chiave `synchronized`, ma quando è opportuno utilizzare questa parola chiave? Vi sono situazioni in cui non è il caso di utilizzarla? Vi sono svantaggi nel suo utilizzo?

Il motivo principale per cui gli sviluppatori non utilizzano la parola chiave `synchronized` è che scrivono programmi a thread unico e codici con un unico scopo. Ad esempio, i compiti legati alla CPU non traggono molti vantaggi dal multithreading: un compilatore non funziona molto meglio se contiene dei thread; il compilatore di Java della Sun non contiene molti metodi `synchronized` e nella maggior parte dei casi assume che l'esecuzione avvenga nel suo thread di controllo senza che sia necessario condividere le risorse con altri thread.

Un altro motivo comune per evitare i metodi synchronized è che sono meno veloci dei metodi non sincronizzati. In semplici test nel JDK versione 1.0.1 della Sun, i metodi synchronized sono stati da tre a quattro volte più lenti delle rispettive controparti non synchronized. Nonostante ciò non significhi che l'intera applicazione risulta tre o quattro volte più lenta, le prestazioni rimangono comunque un problema.

Alcuni programmi necessitano che le prestazioni siano sempre ai massimi livelli, pertanto in questa situazione potrebbe essere appropriato evitare il calo di prestazione associato ai metodi synchronized.

Classi interne di Java 1.1

Con la versione 1.1 sono ora supportate le *classi interne*, presentate nel Capitolo 4. Le classi interne sono classi dichiarate all'interno di un'altra classe. Un'istanza di una classe interna ha un rapporto speciale con l'istanza della classe esterna, infatti le è permesso l'accesso a tutti i campi di tale istanza, inclusi quelli private; inoltre, l'istanza di una classe interna può modificare le variabili della sua istanza esterna.

Le classi interne presentano potenziali problemi per i programmi che utilizzano i thread. Si consideri il seguente esempio di una classe interna e di una classe esterna:

```
public class Esterna {
  private int varPrivata = 1;
  public synchronized void update() {
    varPrivata = varPrivata + 1;
  }
  public class Interna {
    public synchronized void update() {
      varPrivata = varPrivata + 1;
    }
  }
}
```

La classe Esterna ha tre campi: una variabile private, un metodo synchronized e una classe Interna. Questo esempio dimostra che le variabili private di un'istanza esterna possono essere aggiornate dai metodi della stessa e anche da quelli dell'istanza interna: sia Esterna che Interna hanno un metodo update() che modifica varPrivata.

La chiamata di un metodo synchronized dall'istanza esterna è un modo sicuro per aggiornare tale istanza:

```
Esterna istanzaEsterna = getIstanzaEsterna();
istanzaEsterna.update();      // SICURO
```

Tuttavia, non è particolarmente sicuro richiamare un metodo synchronized da un'istanza interna per aggiornare l'istanza esterna:

```
Esterna istanzaEsterna = getIstanzaEsterna();
Esterna.Interna istanzaInterna = istanzaEsterna.new Interna();
istanzaInterna.update();      // NON SICURO
```

Questo secondo codice non è sicuro, in quanto l'utilizzo delle classi interne coinvolge più oggetti e di conseguenza più monitor. Quando viene richiamato `istanzaInterna.update()`, l'unico monitor acquisito è quello della `istanzaEsterna`. Il monitor della `istanzaEsterna` non viene mai acquisito, neanche se viene aggiornata la sua `varPrivata`. Per rendere sicuro il codice, il metodo `update()` della classe `Interna` dovrebbe acquisire il monitor dell'istanza esterna, come indicato di seguito:

```
class Interna {
  public synchronized void update() {
    synchronized (Esterna.this) {      // deve anche bloccare l'istanza esterna
      varPrivata = varPrivata + 1;
    }
  }
}
```

In generale, le classi interne che accedono a variabili di un'istanza esterna dovrebbero utilizzare l'enunciato `synchronized` per bloccare l'istanza esterna durante l'accesso alle variabili. La forma generalizzata per questo tipo di blocco è:

```
synchronized (nome-della-classe-esterna.this) {
  // qui va il codice
}
```

Deadlock

Chiamati anche *abbraccio mortale*, i *deadlock* rappresentano una delle situazioni peggiori che si possono verificare in un ambiente multithreading. I programmi di Java non sono immuni dai deadlock e i programmatori devono assicurarsi di evitarli.

Un deadlock è una situazione in cui due o più thread rimano *in sospeso*, cioè non sono in grado di proseguire. Nel caso più semplice, due thread cercano di acquisire un monitor già in possesso dell'altro. Entrambi vengono tenuti in sospeso, in attesa che diventi disponibile il monitor desiderato, ma i monitor non si libereranno mai: il primo thread aspetta il monitor in possesso del secondo, che aspetta il monitor in possesso del primo; poiché ognuno dei due thread è in attesa dell'altro, nessuno rilascia il suo monitor.

L'applicazione di esempio del Listato 5.5 dà un'idea di come si genera un deadlock.

Listato 5.5 *Un deadlock.*

```
public class Deadlock implements Runnable {
  public static void main(String[] args) {
        Deadlock d1 = new Deadlock();
        Deadlock d2 = new Deadlock();
        Thread t1 = new Thread(d1);
        Thread t2 = new Thread(d2);
        d1.grabIt = d2;
        d2.grabIt = d1;
        t1.start();
        t2.start();
        try { t1.join(); t2.join(); } catch(InterruptedException e) { }
        System.exit(0);
  }
```

```
    Deadlock grabIt;
    public synchronized void run() {
          try { Thread.sleep(2000); } catch(InterruptedException e) { }
          grabIt.sync_method();
    }
    public synchronized void sync_method() {
          try { Thread.sleep(2000); } catch(InterruptedException e) { }
          System.out.println("in sync_method");
    }
}
```

In questa classe, `main()` avvia due thread, ognuno dei quali richiama il metodo `synchronized` `run()` su un `Deadlock`. Quando il primo riprende, cerca di richiamare `sync_method()` dell'altro oggetto `Deadlock`; ovviamente, il monitor del secondo `Deadlock` è in possesso del secondo thread, pertanto il primo thread inizia ad aspettare il monitor. Quando il secondo thread riprende l'esecuzione, cerca di richiamare `sync_method()` del primo oggetto `Deadlock`. Poiché il monitor di questo `Deadlock` è già in possesso del primo thread, il secondo inizia l'attesa. Poiché i thread si aspettano a vicenda, nessuno riprende mai l'esecuzione.

Se si esegue l'applicazione indicata nel Listato 5.5, si nota che non termina mai, cosa comprensibile, in quanto si tratta di un deadlock. Per sapere che cosa avviene veramente all'interno della macchina virtuale, esiste un trucco da utilizzare con il JDK Solaris/Unix per visualizzare lo stato di tutti i thread e di tutti i monitor: premendo Ctrl + \ *nella finestra del terminale in cui viene eseguita l'applicazione di Java, viene inviato un segnale alla macchina virtuale che ne rileva lo stato. Di seguito è presentata una parte del contenuto dello schermo diversi secondi dopo aver avviato l'applicazione con il deadlock:*

```
Deadlock@EE300840/EE334C20 (key=0xee300840):      monitor owner: "Thread-5"
    Waiting to enter:
        "Thread-4"
Deadlock@EE300838/EE334C18 (key=0xee300838):      monitor owner: "Thread-4"
    Waiting to enter:
        "Thread-5"
```

Sono disponibili diversi algoritmi per prevenire e rilevare situazioni di deadlock, ma questo è un argomento che va oltre l'ambito di questo capitolo; molti testi che trattano di database e di sistemi operativi discutono in dettaglio il rilevamento dei deadlock. Sfortunatamente, la macchina virtuale di Java non esegue nessun rilevamento o notifica dei deadlock. Tuttavia, poiché non vi è nulla che le impedisce di farlo, questo comportamento potrebbe essere aggiunto nelle future versioni.

Utilizzo di volatile

Vale la pena menzionare che la parola `volatile` in Java è supportata come modificatore di variabili. Le specifiche del linguaggio indicano che il qualificatore `volatile` indica al compilatore di generare `load` e `store` a ogni accesso a un attributo, anziché memorizzare il valore in un registro. Lo scopo della parola chiave `volatile` è fornire l'accesso sicuro a un attributo, ma questo obiettivo non viene raggiunto.

Nella macchina virtuale del JDK 1.x, la parola chiave `volatile` è ignorata. Non è chiaro se sia stata abbandonata a favore dei monitor e dei metodi `synchronized` o se sia stata inclusa solo per ottenere un aspetto più simile al C/C++. Indipendentemente da tutto ciò, la parola chiave `volatile` non ha alcuna utilità ed al suo posto si utilizzano i metodi `synchronized`.

Sincronizzazione

Dopo aver appreso come vengono utilizzati i metodi `synchronized` per rendere i programmi sicuri rispetto ai thread, ci si potrebbe chiedere quale sia il vantaggio dei monitor. I monitor non solo permettono di bloccare gli oggetti, ma vengono utilizzati anche per coordinare diversi thread utilizzando i metodi `wait()` e `notify()` disponibili in ogni oggetto di Java.

La necessità di coordinare i thread

In un programma Java, i thread sono spesso interdipendenti, vale a dire che per completare un'operazione o per esaudire una richiesta un thread può dipendere da un altro. Ad esempio, un programma di foglio elettronico può eseguire un ricalcolo in un thread separato. Se un thread dell'interfaccia utente cerca di aggiornare la visualizzazione del foglio elettronico, dovrebbe coordinarsi con quello del ricalcolo, avviando l'aggiornamento dello schermo solo dopo che quest'ultimo ha terminato la sua operazione.

Vi sono molte altre situazioni in cui è utile coordinare due o più thread. Nel seguente elenco vengono presentate solo alcune delle possibilità.

- ✔ **Si utilizzano spesso buffer condivisi per comunicare dati tra i thread.** In questo scenario, un thread scrive in un buffer condiviso e un altro legge dal buffer. Quando il thread di lettura cerca di leggere dal buffer, dovrebbe coordinarsi con il thread di scrittura, richiamando i dati dal buffer condiviso solo dopo che il thread di scrittura ve li ha inseriti. Se il buffer è vuoto, il thread di lettura deve rimanere in attesa di dati. Quando il thread di scrittura ha completato il riempimento del buffer, lo notifica al thread di lettura, in modo che questo possa procedere.
- ✔ **Molti thread devono eseguire un'azione identica, ad esempio caricare un file di immagine in una rete.** Questi thread possono ridurre il carico generale del sistema, se uno solo esegue il compito mentre gli altri aspettano che il lavoro sia completato. I thread in attesa devono aspettare senza occupare il tempo della CPU, passando temporaneamente allo stato `NOT RUNNABLE`, come discusso più avanti in questo capitolo. Questo è esattamente il modello utilizzato nella classe `java.awt.MediaTracker`.

Negli esempi precedenti sono state utilizzate ripetutamente le parole "aspettare" e "notificare". Queste parole esprimono due concetti fondamentali per il coordinamento dei thread: un thread *aspetta* che si verifichi una condizione o un evento, mentre si *notifica* a un thread in attesa che si è verificata una condizione o un evento. La traduzione inglese di queste parole è utilizzata in Java anche per i nomi dei metodi da chiamare per coordinare i thread: `wait()` e `notify()`, nella classe `Object`.

Come indicato precedentemente in questo capitolo, a ogni oggetto di Java è associato un monitor, cosa utile a questo punto, in quanto i monitor vengono utilizzati per implementare le primitive di coordinamento dei thread di Java. Nonostante i monitor non siano direttamente visibili ai programmatori, nella classe `Object` vi è un'API che permette di interagire con il monitor di un oggetto. L'API consiste di due metodi: `wait()` e `notify()`.

Condizioni, wait() e notify()

I thread normalmente vengono coordinati utilizzando *condizioni* o *variabili condizionali*. Una condizione è un'istruzione logica che deve rimanere `true` (vera) affinché un thread prosegua; diversamente, per proseguire il thread deve aspettare che lo diventi. In Java questo viene di norma espresso nel seguente modo:

```
while ( ! la_condizione_che_aspetto ) {
    wait();
}
```

Innanzitutto si controlla se la condizione desiderata è già vera, nel qual caso non è necessario aspettare. Se la condizione non è ancora vera, si richiama il metodo `wait()`. Quando questo termina, si controlla nuovamente la condizione per assicurarsi che sia diventata vera.

Richiamando `wait()` su un oggetto, si pone in pausa il thread corrente e lo si aggiunge alla *coda di attesa delle variabili condizionali* del monitor dell'oggetto. Questa coda contiene un elenco di tutti i thread attualmente bloccati all'interno di `wait()` su quell'oggetto. Il thread non viene rimosso dall'elenco finché un thread diverso non richiama `notify()` su quell'oggetto. Così facendo viene ripresa l'esecuzione di un solo thread di quelli in attesa, informandolo che è cambiata una condizione dell'oggetto.

Esistono altre due varianti del metodo `wait()`: la prima utilizza un unico parametro, un tempo limite in millesimi di secondo, la seconda utilizza due parametri, un tempo limite più preciso, espresso in millisecondi e in nanosecondi. Questi metodi vengono utilizzati quando non si vuole aspettare un evento all'infinito. Se si desidera abbandonare lo stato di attesa dopo un determinato periodo di tempo (definito come *time-out*), si utilizza uno dei seguenti metodi:

```
wait(long millisecondi);
wait(long millisecondi, int nanosecondi);
```

Sfortunatamente, questi metodi non permettono di determinare in che modo è terminato `wait()`: se è stato eseguito un `notify()` o se il metodo ha esaurito il suo tempo. Questo comunque non è un grosso problema, in quanto è possibile controllare nuovamente la condizione di attesa e il tempo del sistema per determinare quale evento si è verificato.

Nelle versioni 1.0.x e 1.1 del JDK, il metodo `wait(int millisecondi, int nanosecondi)` utilizza il parametro `nanosecondi` per arrotondare il parametro `millisecondi` al millesimo di secondo più prossimo. Non è ancora supportata l'attesa in nanosecondi.

I metodi `wait()` e `notify()` devono essere richiamati all'interno di un metodo `synchronized` o di un enunciato `synchronized`, come discusso più dettagliatamente nel paragrafo: "Possesso dei monitor" più avanti in questo capitolo.

Un esempio di coordinamento dei thread

Un esempio classico di coordinamento dei thread utilizzato in molti testi di informatica è quello del *buffer limitato*, che utilizza un buffer a dimensione di memoria fissa per le comunicazioni tra due processi o tra due thread. Per risolvere questo problema, è necessario coordinare il thread di lettura e quello di scrittura in modo che vengano soddisfatte le seguenti condizioni.

- ✔ Quando il thread di scrittura cerca di scrivere in un buffer pieno, il thread viene sospeso finché non vengono eliminati degli elementi dal buffer.
- ✔ Quando il thread di lettura elimina gli elementi dal buffer pieno, il thread di scrittura viene informato del cambiamento della condizione del buffer e prosegue la scrittura.
- ✔ Quando il thread di lettura cerca di leggere da un buffer vuoto, il thread viene sospeso finché non vengono aggiunti degli elementi al buffer.
- ✔ Quando il thread di scrittura aggiunge degli elementi a un buffer vuoto, il thread di lettura viene informato del cambiamento di condizione del buffer e prosegue la lettura.

Il seguente listato mostra un'implementazione in Java del problema del buffer limitato. In questo esempio vi sono tre classi principali: la classe `Produttore`, la classe `Consumatore` e la classe `Buffer`. Si analizzi per prima la classe `Produttore`:

```
public class Produttore implements Runnable {
  private Buffer buffer;
  public Produttore(Buffer b) {
        buffer = b;
  }
  public void run() {
      for (int i=0; i<250; i++) {
          buffer.put((char)('A' + (i%26)));    // scrive sul buffer
      }
  }
}
```

La classe `Produttore` implementa l'interfaccia `Runnable`, il che suggerisce che verrà utilizzata in un thread. Quando viene richiamato il metodo `run()` di `Produttore`, in un buffer vengono scritti in rapida successione 250 caratteri.

La classe `Consumatore` è semplice come la classe `Produttore`:

```
public class Consumatore implements Runnable {
  private Buffer buffer;
  public Consumatore(Buffer b) {
        buffer = b;
  }
  public void run() {
```

```
        for (int i=0; i<250; i++) {
          System.out.println(buffer.get());      // legge dal buffer
        }
    }
  }
```

Anche questa classe è un'interfaccia Runnable. Il suo metodo run() legge 250 caratteri da un buffer.

La classe Buffer è già stata menzionata, compresi due dei suoi metodi: put(char) e get(). Il Listato 5.6 contiene l'intera classe Buffer.

Listato 5.6 *La classe Buffer.*

```
public class Buffer {
  private char[] buf;     // memoria del buffer
  private int last;       // ultima posizione occupata
  public Buffer(int sz) {
        buf = new char[sz];
        last = 0;
  }
  public boolean isFull()  { return (last == buf.length); }
  public boolean isEmpty() { return (last == 0);          }
  public synchronized void put(char c) {
        while(isFull()) {                     // attende che vi sia spazio disponibile
          try { wait(); } catch(InterruptedException e) { }
        }
        buf[last++] = c;
        notify();
  }
  public synchronized char get() {
        while(isEmpty()) {                    // attende qualcosa da scrivere
          try { wait(); } catch(InterruptedException e) { }
        }
        char c =  buf[0];
        System.arraycopy(buf, 1, buf, 0, —last);
        notify();
        return c;
  }
}
```

Quando si utilizzano wait() e notify() per la prima volta, si nota una contraddizione. Questi metodi devono essere richiamati da metodi synchronized, pertanto se wait() viene richiamato all'interno di un metodo sincronizzato, come può un altro thread entrare in un metodo sincronizzato per richiamare notify()? Il thread in attesa non possiede il monitor dell'oggetto, impedendo ad altri thread di entrare nel metodo synchronized?
La risposta a questo paradosso è che wait() rilascia temporaneamente il possesso del monitor dell'oggetto; prima che possa essere restituito, wait() deve tuttavia riacquistare il possesso del monitor. Rilasciando il monitor, il metodo wait() permette ad altri thread di acquisirlo e pertanto di richiamare notify().

La classe `Buffer` è semplicemente un buffer di memorizzazione. È possibile utilizzare `put()` per inserirvi degli elementi, in questo caso caratteri, e `get()` per richiamare le voci dal buffer.

Si noti l'utilizzo di `wait()` e di `notify()` in questi metodi. Nel metodo `put()` viene eseguito un `wait()` mentre il buffer è pieno, il che rende impossibile l'aggiunta di altri elementi. Al termine del metodo `get()`, richiamando `notify()` si garantisce che vengano attivati tutti i thread in attesa nel metodo `put()`, permettendo così di continuare ad aggiungere elementi al buffer. In modo simile, nel metodo `get()` viene eseguito un `wait()`, se il buffer è vuoto, facendo in modo che nessun elemento possa essere eliminato dal buffer.

Il metodo `put()` richiama `notify()` per garantire che venga ripresa l'esecuzione di tutti i thread in attesa in `get()`.

In Java sono incluse due classi simili alla classe `Buffer` presentata in questo esempio. Queste classi, `java.io.PipedOutputStream` e `java.io.PipedInputStream`, sono utili per comunicare flussi di dati tra i thread. Se si decomprime il file `src.zip` incluso nel JDK, è possibile esaminare queste classi per vedere come gestiscono il coordinamento tra i thread.

Coordinamento avanzato dei thread

I metodi `wait()` e `notify()` semplificano il compito di coordinamento di diversi thread in un programma concorrente di Java. Tuttavia, per utilizzare appieno questi metodi, è necessario comprendere alcuni altri dettagli. I seguenti paragrafi forniscono ulteriori informazioni sul coordinamento dei thread in Java.

Possesso dei monitor

I metodi `wait()` e `notify()` sono sottoposti a una restrizione: possono essere richiamati solo quando il thread corrente possiede il monitor dell'oggetto. Nella maggior parte dei casi, `wait()` e `notify()` vengono richiamati dall'interno di un metodo `synchronized`, come nel seguente esempio:

```
public synchronized void method() {
    ...
    while (!condition) {
      wait();
    }
    ...
}
```

In questo caso, il modificatore `synchronized` garantisce che il thread che richiama `wait()` possiede già il monitor al momento della chiamata.

Se si cerca di richiamare `wait()` o `notify()` senza aver prima acquisito il possesso del monitor dell'oggetto, ad esempio da un metodo non sincronizzato, la macchina virtuale genera un'eccezione `IllegalMonitorStateException`. Il seguente esempio mostra che cosa succede se si richiama `wait()` senza aver prima acquisito il possesso del monitor:

I monitor e l'istruzione synchronized

Tutti gli oggetti di Java possono partecipare alla sincronizzazione dei thread utilizzando i metodi `wait()` e `notify()`. Tuttavia, il requisito del "possesso del monitor" introduce un cavillo per alcuni tipi di oggetti, ad esempio per gli array (i tipi di array in Java derivano dalla classe `java.lang.Object`, in cui sono definiti i metodi `wait()` e `notify()`). I metodi `wait()` e `notify()` possono essere richiamati su oggetti array in Java, ma il possesso del monitor deve essere stabilito utilizzando l'istruzione `synchronized` e non un metodo sincronizzato. Il seguente codice mostra l'utilizzo dei monitor applicato a un array di Java:

```
// attende un evento su questo array
Object[] array = getArray();
synchronized (array) {
  array.wait();
}
...
// notifica i thread in attesa
Object[] array = getArray();
synchronized (array) {
  array.notify();
}
```

```
public class NonOwnerTest {
  public static void main(String[] args) {
        NonOwnerTest not = new NonOwnerTest();
        not.method();
  }
  public void method() {
        try { wait(); } catch(InterruptedException e) { }    // Brutta cosa!
  }
}
```

Se si esegue questa applicazione di Java, nel terminale viene stampato il seguente testo:

```
java.lang.IllegalMonitorStateException: current thread not owner
        at java.lang.Object.wait(Object.java)
        at NonOwnerTest.method(NonOwnerTest.java:10)
        at NonOwnerTest.main(NonOwnerTest.java:5)
```

Quando si richiama il metodo `wait()` su un oggetto, se si desidera evitare questa eccezione è necessario avere il possesso del relativo monitor.

Diversi thread in attesa

È possibile che diversi thread siano in attesa dello stesso oggetto. Ciò può accadere quando diversi thread sono in attesa dello stesso evento. Ad esempio, tornando alla classe `Buffer` descritta precedentemente, il buffer veniva attivato da un singolo `Produttore` e da un singolo `Consumatore`. Che cosa succederebbe se vi fossero diversi `Produttore`?

Se il buffer si riempisse, diversi Produttore potrebbero tentare di utilizzare put() per aggiungervi elementi e si bloccherebbero tutti nel metodo put(), aspettando che arrivi un Consumatore per liberare dello spazio nel buffer.

Quando si richiama notify(), potrebbero esservi zero, uno o più thread bloccati in un metodo wait() nel monitor. Se non vi sono thread in attesa, la chiamata a notify() viene detta *no-op*, che significa che non influisce sugli altri thread. Se vi è un unico thread in wait(), questo viene informato e inizia ad aspettare che il monitor venga rilasciato dal thread che ha effettuato la chiamata a notify(). Se in un wait() vi sono due o più thread, la macchina virtuale sceglie un singolo thread in attesa e lo informa. Il metodo utilizzato per scegliere un thread varia da piattaforma a piattaforma, pertanto i programmi non dovrebbero basarsi sulla VM per selezionare un thread specifico tra quelli in attesa.

Utilizzo di notifyAll()

In determinate situazioni, è possibile informare tutti i thread in attesa di un oggetto. L'API Object include un metodo che permette di farlo: notifyAll(). Il metodo notify() ripristina l'esecuzione di un singolo thread in attesa, mentre il metodo notifyAll() ripristina l'esecuzione di tutti i thread in attesa di un oggetto.

Quando occorre utilizzare notifyAll()? Si consideri la classe java.awt.MediaTracker, utilizzata per registrare lo stato delle immagini che vengono caricate in una rete. Diversi thread potrebbero essere in attesa dello stesso oggetto MediaTracker, aspettando che tutte le immagini siano caricate. Quando il MediaTracker rileva che tutte le immagini sono state caricate, viene richiamato notifyAll() per comunicarlo a tutti i thread in attesa. Viene utilizzato notifyAll() perché il MediaTracker non sa quanti thread sono in attesa; se venisse utilizzato notify(), alcuni thread in attesa potrebbero non ricevere l'informazione che è stato terminato il trasferimento e rimarrebbero in attesa, probabilmente tenendo in sospeso l'intero applet.

notifyAll() può essere utilizzato anche nel Listato 5.6, presentato precedentemente in questo capitolo. In questo codice, la classe Buffer utilizzava il metodo notify() per informare un singolo thread in attesa di un buffer pieno o vuoto. Tuttavia, non vi erano garanzie che vi fosse un solo thread in attesa. Il Listato 5.7 mostra una versione modificata della classe Buffer, chiamata Buffer2, che utilizza notifyAll().

Listato 5.7 *La classe Buffer2 che utilizza notifyAll().*

```
public class Buffer2 {
  private char[] buf;                 // memoria
  private int last = 0;               // ultima posizione occupata
  private int writers_waiting = 0;    // numero di thread in attesa in put()
  private int readers_waiting = 0;    // numero di thread in attesa in get()
  public Buffer2(int sz) {
        buf = new char[sz];
}
  public boolean isFull()  { return (last == buf.length); }
  public boolean isEmpty() { return (last == 0);          }
```

```
    public synchronized void put(char c) {
          while(isFull()) {
            try     { writers_waiting++;  wait(); }
            catch   (InterruptedException e) { }
            finally { writers_waiting–; }
          }
          buf[last++] = c;
          if (readers_waiting > 0) {
            notifyAll();
          }
    }
    public synchronized char get() {
          while(isEmpty()) {
            try     { readers_waiting++;  wait(); }
            catch   (InterruptedException e) { }
            finally { readers_waiting–; }
          }
          char c =  buf[0];
          System.arraycopy(buf, 1, buf, 0, –last);
          if (writers_waiting > 0) {
            notifyAll();
          }
          return c;
    }
}
```

I metodi `get()` e `put()` sono stati resi più intelligenti: ora controllano se è necessario comunicare delle informazioni e quindi utilizzano `notifyAll()` per informare di un evento tutti i thread in attesa.

Riepilogo

Questo capitolo ha presentato la programmazione multithreading in Java. Tra le altre cose, sono stati trattati i seguenti argomenti.

- ✔ Creazione di classi di thread personalizzate con sottoclassi di `Thread` o implementando `Runnable`.
- ✔ Utilizzo della classe `ThreadGroup` per gestire gruppi di thread.
- ✔ Stati e gestione dei thread.
- ✔ Creazione di classi sicure rispetto ai thread utilizzando la parola chiave `synchronized` per proteggere gli oggetti da modifiche simultanee.
- ✔ Influenza dei monitor sulla programmazione concorrente in Java.
- ✔ Coordinamento delle azioni di diversi thread richiamando i metodi `wait()` e `notify()`.

Non è difficile utilizzare i thread di Java; dopo aver letto questo capitolo, si dovrebbe iniziare a vedere come sia possibile impiegarli per migliorare la programmazione.